

CodeFlow Catalyst - Demo Scenarios

Demo Flow (Total: 10 minutes)

Scenario 1: The Fee Calculation Change (3 minutes)

Story: A developer needs to update international transfer fees.

Setup:

- Show `PaymentProcessor.java` in the editor.
- Point to the `calculateFee()` method (line 58).

Change:

```
// OLD CODE:  
return amount * 0.03; // 3% fee  
  
// NEW CODE (change to):  
return amount * 0.025 + 15.0; // 2.5% + $15 flat fee
```

Expected Analysis:

- **Risk Score:** 7.5/10 (HIGH)
- **Direct Dependencies:** 4
 - `FraudDetection` (uses fee in calculations)
 - `AccountBalance` (deducts fee from balance)
 - `TransactionDAO` (stores fee)
 - `NotificationService` (displays fee in notification)
- **Indirect Dependencies:** 2
 - `RegulatoryReporting` (fee disclosure requirements)
 - `TRANSACTIONS` table (fee column affected)

AI Insights:

- "Fee calculation change affects customer-facing displays"
- "Regulatory concern: Fee disclosure (Regulation E) - 30 day notice required"
- "Similar change in March 2024 caused reconciliation issues"

Demo Actions:

1. Show the code in the editor.
2. Make the change.
3. Commit to GitHub.
4. Show that the webhook triggers the analysis.
5. Show the dashboard updating in real-time.

6. Walk through the dependency graph.
7. Explain the AI insights.
8. Show the test plan generated.

Key Points to Emphasize:

- Found a hidden dependency (RegulatoryReporting).
- AI caught a compliance requirement.
- Historical incident was referenced.
- Automated test scenarios were created.

Key Takeaway: CodeFlow caught a hidden *compliance* and *financial* risk, not just a *code bug*.

Scenario 2: Database Schema Change (2 minutes)

Story: Add a new column to the TRANSACTIONS table.

Change:

```
ALTER TABLE TRANSACTIONS
ADD COLUMN currency VARCHAR(3) DEFAULT 'USD';
```

Expected Analysis:

- **Risk Score:** 8.2/10 (CRITICAL)
- **Affected Code Files:** 3
 - TransactionDAO.java (INSERT/SELECT statements)
 - AccountBalance.java (reads TRANSACTIONS)
 - RegulatoryReporting.java (generates reports from table)

AI Insights:

- "Schema change affects 3 modules with hard-coded column lists."
- " SELECT * statements will break in TransactionDAO ."
- "Data migration required for 15 million existing rows."

Demo Actions:

1. Show the SQL change.
2. Trigger the analysis.
3. Show which Java files need updates.
4. Highlight the data migration complexity.

Key Takeaway: The analysis instantly bridged the gap between a database change and the application code, preventing runtime errors and saving hours of manual detective work.

Scenario 3: The Domino Effect (2 minutes)

Story: Refactor the FraudDetection.checkTransaction() method signature.

Change:

```
// OLD:  
public boolean checkTransaction(Payment payment)  
  
// NEW:  
public FraudResult checkTransaction(Payment payment, Context context)
```

Expected Analysis:

- **Risk Score:** 9.1/10 (CRITICAL)
- Breaking Change Detected
- **Immediate Impact:** PaymentProcessor.java (line 35 - method call breaks)
- **Cascade Impact:** All payment flows affected.

Dependency Chain:

```
FraudDetection.java (YOU CHANGED)  
↓  
PaymentProcessor.java (BREAKS - method signature changed)  
↓  
MobileApp API (BREAKS - payment processing fails)  
↓  
5 million users affected
```

AI Insights:

- "CRITICAL: This is a breaking API change."
- "Recommendation: Use method overloading for backward compatibility."
- "Estimated impact: 2 hours to fix all calling code."

Demo Actions:

1. Show the method signature change.
2. Explain the concept of a "breaking change."
3. Show the cascade of impacts.
4. Display the AI recommendation.

Key Takeaway: The system immediately flagged a breaking API change and mapped the full "domino effect," protecting millions of end-users from a critical failure.

Scenario 4: The Saved Disaster (3 minutes)

Story: A developer attempts a deployment on a Friday at 5 PM.

Change:

```
// In PaymentProcessor.java, line 58  
// Developer adds date manipulation  
  
private double calculateFee(double amount, String type) {
```

```
// NEW CODE: Apply month-end surcharge
LocalDate today = LocalDate.now();
if (today.getDayOfMonth() == 31) {
    amount = amount * 1.1; // 10% surcharge on last day
}

// ... rest of fee calculation
}
```

Expected Analysis:

- **Risk Score:** 9.5/10 (CRITICAL)
- **Deployment Timing Risk:** 2.0x multiplier (Friday evening)
- Historical Pattern Match Found

AI Insights:

- "⚠️ CRITICAL: Deploying on Friday at 5 PM."
- "Edge case detected: What about months with 28/29/30 days?"
- "Similar date logic bug in 2023 caused month-end processing failure."
- "Recommendation: DELAY deployment until Monday morning."
- "Test specifically with: Feb 28, Feb 29, April 30."

Demo Actions:

1. Show the developer trying to commit at 5 PM on a Friday.
2. Show the system flagging the high risk.
3. Show the AI catching the date edge case.
4. Show the historical incident reference.
5. Show the recommendation to postpone deployment.

Key Message:

"This analysis just prevented a weekend disaster! The developer would have gone home, and the bug would have hit production on month-end close."

Demo Preparation Checklist

Before Demo (Day 13):

- [] Neo4j populated with sample data
- [] Backend server running (localhost:8000)
- [] Frontend dashboard running (localhost:3000)
- [] GitHub webhook configured (or use simulation)
- [] All 4 demo scenarios tested
- [] Screenshots/recordings as backup
- [] Presentation slides ready

Demo Environment:

```
# Terminal 1: Backend logs
cd backend && unicorn app.main:app --reload --port 8000

# Terminal 2: Frontend
cd frontend && npm start

# Terminal 3: Neo4j Browser
open http://localhost:7474

# Browser: Dashboard
open http://localhost:3000
```

Backup Plan:

If the live demo fails:

- Video recording of a working demo
- Pre-generated analysis results in the database
- Static screenshots of each step
- Slide deck with embedded images

Talking Points for Each Scenario

Scenario 1 (Fee Change):

- "Here, we see it found the hidden *regulatory* impact, something a human code review would almost certainly miss."
- "The AI referenced a similar incident from our codebase's history."

Scenario 2 (Database):

- "Database changes are especially tricky and high-risk."
- "It found all the Java files that were using hard-coded column lists."
- "This analysis just saved the database and app teams *hours* of detective work trying to figure out why the application was suddenly crashing."

Scenario 3 (Breaking Change):

- "This is a classic 'breaking API change'."
- "Watch how the impact cascades through the system, all the way to the end-user."
- "5 million mobile users would have been affected."

Scenario 4 (Saved Disaster):

- "This scenario is based on a real incident at a major bank."
- "The AI caught a tricky date-logic edge case that humans often miss."
- "The deployment timing analysis prevented a weekend-long incident."

Audience Questions & Answers

Q: How does it handle false positives? A: Users can provide feedback on the analysis, and the system learns to improve its accuracy over time. After 6 months, we typically see accuracy rates reach 95%+.

Q: What about proprietary code security? A: The entire platform can be run completely on-premise. The AI models can be self-hosted, so your code never leaves your infrastructure.

Q: How long does an analysis take? A: A typical analysis for a PR takes 30-60 seconds. For very large, complex codebases, it might take 2-3 minutes. This is much faster than the 4-6 hour manual process it replaces.

Q: Does it work with other languages? A: This demo is in Java, but the underlying dependency engine supports 10+ languages. It's built to be extensible.

Q: How does it integrate with CI/CD? A: It integrates via webhooks with GitHub, GitLab, and Bitbucket. You can configure it to automatically block risky PRs from being merged.

Q: What is the cost? A: The platform is built on a powerful open-source core, making it easy to get started. We offer flexible enterprise plans that add features like priority support, self-hosted AI models, and advanced security integrations.